Phantom-GRAPE: numerical software library to accelerate collisionless N-body simulation with SIMD instruction set on x86 architecture

Ataru Tanikawa [a,*], Kohji Yoshikawa [a], Keigo Nitadori [a] & Takashi Okamoto [a]

[a] *Center for Computational Science, University of Tsukuba, 1–1–1, Tennodai, Tsukuba, Ibaraki 305–8577, Japan*

Abstract

We have developed a numerical software library for collisionless N-body simulations named "Phantom-GRAPE" which highly accelerates force calculations among particles by use of a new SIMD instruction set extension to the x86 architecture, Advanced Vector eXtensions (AVX), an enhanced version of the Streaming SIMD Extensions (SSE). In our library, not only the Newton's forces, but also central forces with an arbitrary shape $f(r)$, which has a finite cutoff radius $r_{\rm cut}$ (i.e. $f(r) = 0$ at $r > r_{\rm cut}$), can be quickly computed. In computing such central forces with an arbitrary force shape $f(r)$, we refer to a pre-calculated look-up table. We also present a new scheme to create the look-up table whose binning is optimal to keep good accuracy in computing forces and whose size is small enough to avoid cache misses. Using an Intel Core i7–2600 processor, we measure the performance of our library for both of the Newton's forces and the arbitrarily shaped central forces. In the case of Newton's forces, we achieve 2×10^9 interactions per second with one processor core (or 75 GFLOPS if we count 38 operations per interaction), which is 20 times higher than the performance of an implementation without any explicit use of SIMD instructions, and 2 times than that with the SSE instructions. With four processor cores, we obtain the performance of 8×10^9 interactions per second (or 300 GFLOPS). In the case of the arbitrarily shaped central forces, we can calculate 1×10^9 and 4×10^9 interactions per second with one and four processor cores, respectively. The performance with one processor core is 6 times and 2 times higher than those of the implementations without any use of SIMD instructions and with the SSE instructions. These performances depend only weakly on the number of particles, irrespective of the force shape. It is good contrast with the fact that the performance of force calculations accelerated by graphics processing units (GPUs) depends strongly on the number of particles. Substantially weak dependence of the performance on the number of particles is suitable to collisionless N-body simulations, since these simulations are usually performed with sophisticated N-body solvers such as Tree- and TreePM-methods combined with an individual timestep scheme. We conclude that collisionless N-body simulations accelerated with our library have significant advantage over those accelerated by GPUs, especially on massively parallel environments.

Key words: Stellar dynamics, Method: N-body simulations

1. Introduction

Self-gravity is one of the most essential physical processes in the universe, and plays important roles in almost all categories of astronomical objects such as globular clusters, galaxies, galaxy clusters, etc. In order to follow the evolution of such systems, gravitational N-body solvers have been widely used in numerical astrophysics.

Due to prohibitively expensive computational cost in directly solving N-body problems, many efforts have been made to reduce it in various ways. For example, several sophisticated algorithms to compute gravitational forces among many particles with reduced computational cost have been developed, such as Tree method (Barnes & Hut, 1986), PPPM method (Hockney & Eastwood, 1981), TreePM method (Xu, 1995), etc.

Another approach is to improve the computational performance with the aid of additional hardware, such as GRAPE (GRAvity PipE) systems, special-purpose accelerators for gravitational N-body simulations (Sugimoto et al., 1990; Makino et al., 2003; Fukushige et al., 2005), and general-purpose computing on Graphics Processing Units (GPGPUs). GRAPE systems have been used for further improvement of existing N-body solvers such as Tree method (Makino, 1991), PPPM method (Brieu et al., 1995; Yoshikawa & Fukushige, 2005), TreePM method (Yoshikawa & Fukushige, 2005), P^2M^2 tree method (Kawai

[*] Corresponding author.
Email address: `tanikawa@ccs.tsukuba.ac.jp` (Ataru Tanikawa).

et al., 2004), and PPPT method (Oshino et al., 2011). They have also adapted to simulation codes for dense stellar systems based on fourth-order Hermite scheme, such as `NBODY4` (Johnson & Aarseth, 2006), `NBODY1` (Harfst et al., 2007), `kira` (Portegies Zwart et al., 2008), and `GORILLA` (Tanikawa & Fukushige, 2009). Recently, Hamada & Iitaka (2007), Portegies Zwart et al. (2007), Gaburov et al. (2009), and Bédorf et al. (2012) explored the capability of commodity graphics processing units (GPUs) as hardware accelerators for N-body simulations and achieved similar to or even higher performance than the GRAPE-6A and GRAPE-DR board.

A different approach to improve the performance of N-body calculations is to utilize Streaming SIMD Extensions (hereafter SSE), a SIMD (Single Instruction, Multiple Data) instruction set implemented on x86 and x86_64 processors. Nitadori et al. (2006) exploited the SSE and SSE2 instruction sets, and achieved speeding up of the Hermite scheme (Makino & Aarseth, 1992) in mixed precision for collisional self-gravitating systems. Although unpublished in literature, Nitadori, Yoshikawa, & Makino have also developed a numerical library for N-body calculations in single-precision for collisionless self-gravitating systems in which two-body relaxation is not physically important and therefore single-precision floating-point arithmetic suffices for the required numerical accuracy. Furthermore, along this approach, they have also improved the performance in computing arbitrarily-shaped forces with a cutoff distance, defined by a user-specified function of inter-particle separation. Such capability to compute force shapes other than Newton's inverse-square gravity is necessary in PPPM, TreePM, and Ewald methods. It should be noted that GRAPE-5 and the later families of GRAPE systems have similar capability to compute the Newton's force multiplied by a user-specified cutoff function (Kawai et al., 2000), and can be used to accelerate PPPM and TreePM methods for cosmological N-body simulations (Yoshikawa & Fukushige, 2005). Based on these achievements, a publicly available software package to improve the performance of both collisional and collisionless N-body simulations has been developed, which was named "Phantom-GRAPE" after the conventional GRAPE system. A set of application programming interfaces of Phantom-GRAPE for collisionless simulations is compatible to that of GRAPE-5. Phantom-GRAPE is widely used in various numerical simulations for galaxy formation (Saitoh et al., 2008, 2009) and the cosmological large-scale structures (Ishiyama et al., 2008, 2009a,b, 2010, 2011).

Recently, a new processor family with "Sandy Bridge" micro-architecture[1] by Intel Corporation and that with "Bulldozer" micro-architecture[2] by AMD Corporation have been released. Both of the processors support a new set of instructions known as Advanced Vector eXtensions (AVX), an enhanced version of the SSE instructions. In the AVX instruction set, the width of the SIMD registers is extended from 128-bit to 256-bit. We can perform SIMD operations on two times larger data than before. Therefore, the performance of a calculation with the AVX instructions should be two times higher than that with the SSE instructions if the execution unit is also extended to 256-bit.

Tanikawa et al. (2012) (hereafter, paper I) developed a software library for *collisional* N-body simulations using the AVX instruction set in the mixed precision, and achieved a fairly high performance. In this paper, we present a similar library implemented with the AVX instruction set but for *collisionless* N-body simulations in single-precision.

The structure of this paper is as follows. In section 2, we overview the AVX instruction set. In section 3, we describe the implementation of Phantom-GRAPE. In section 4 and 5, we show the accuracy and performance, respectively. In section 6, we summarize this paper.

2. The AVX instruction set

In this section, we present a brief review of the Advanced Vector eXtensions (AVX) instruction set. Details of the difference between SSE and AVX is described in section 3.1 of paper I. AVX is a SIMD instruction set as well as SSE, and supports many operations, such as addition, subtraction, multiplication, division, square-root, approximate inverse-square-root, several bitwise operations, etc. In such operations, dedicated registers with 256-bit length called "YMM registers" are used to store the eight single-precision floating-point numbers or four double-precision floating-point numbers. Note that the lower 128-bit of the YMM registers have alias name "XMM registers", and can be used as the dedicated registers for the SSE instructions for a backward compatibility.

An important feature of AVX and SSE instruction sets is the fact that they have a special instruction for a very fast approximation of inverse-square-root with an accuracy of about 12-bit. Actually, this instruction is quite essential to improve the performance of the gravitational force calculations, since the most expensive part in the force calculation is an execution of inverse-square-root of squared distances of the particle pairs. As already discussed in Nitadori et al. (2006), the approximate values can be adopted as initial values of the Newton-Raphson iteration to improve the accuracy, and we can obtain 24-bit accuracy after one Newton-Raphson iteration. For collisionless self-gravitating systems, however, the accuracy of $\simeq 12$ bits is sufficient because the accuracy of inverse-square-root does not affect the resultant force accuracy if one adopts an approximate N-body solver such as Tree, PPPM and TreePM methods. Therefore, we use the raw approximate instruction throughout this study.

Since the present-day compilers cannot always detect concurrency of the loops effectively, and cannot fully resolve

[1] `http://www.intel.com/content/dam/doc/manual/`
[2] `http://support.amd.com/us/Processor_TechDocs/`

the mutual dependency among data in the code, it is quite rare that compilers generate codes with SIMD instructions in effective manners from codes expressed in high-level languages. For an efficient use of the AVX instructions, we need to program with assembly-languages explicitly or compiler-dependent intrinsic functions and data type extensions. In assembly-languages, we can manually control the assignment of YMM registers to computational data, and minimize the access to the main memory by optimizing the assignment of each register. In this work, we adopt an implementation of the AVX instructions using inline-assembly language with C expression operands, embedded in C-language, which is a part of language extensions of GCC (GNU Compiler Collection).

3. Implementation

Here, we describe the detailed implementation to accelerate N-body calculation using the AVX instructions. For a given set of positions $\boldsymbol{r}_i$ of N particles, we try to accelerate the calculations of a gravitational force given as follows:

$$\boldsymbol{a}_i = \sum_{j=1}^{N} \frac{Gm_j(\boldsymbol{r}_j - \boldsymbol{r}_i)}{(|\boldsymbol{r}_j - \boldsymbol{r}_i|^2 + \epsilon^2)^{3/2}}, \tag{1}$$

where G is the gravitational constant, m_j the mass of the j-th particle, and ϵ the gravitational softening length. In addition to that, we also try to accelerate the computations of central forces among particles with an arbitrary force shape $f(r)$ given by

$$\boldsymbol{a}_i = \sum_{j=1}^{N} m_j f(|\boldsymbol{r}_j - \boldsymbol{r}_i|) \frac{\boldsymbol{r}_j - \boldsymbol{r}_i}{|\boldsymbol{r}_j - \boldsymbol{r}_i|}, \tag{2}$$

where $f(r)$ specifies the shape of the force as a function of inter-particle separation r with a cutoff distance $r_{\rm cut}$ (i.e. $f(r) = 0$ at $r > r_{\rm cut}$). In the above expressions, particles with subscript "j" exert forces on those with subscript "i". In the rest of this paper, the former are referred to as "j-particles", and the latter as "i-particles" just for convenience.

Since individual forces exerted by j-particles on i-particles can be computed independently, we can calculate forces exerted by multiple j-particles on multiple i-particles in parallel. As described in the previous section, the AVX instructions can execute operations of eight single-precision floating-point numbers on YMM registers in parallel. By utilizing this feature of the AVX instructions, the forces on four i-particles from two j-particles can be computed simultaneously in a SIMD manner.

3.1. *Structures for the particle data*

In computing the forces on four i-particles from two j-particles, we assign the data of i- and j-particles to YMM registers in the way shown in Figure 1. Suppose that a and b in Figure 1 are x-components of i- and j-particles, respectively. Subtracting data in the YMM register (1) of Figure 1 from data in the YMM register (2) of Figure 1, we simultaneously obtain x-components of eight relative positions c in the YMM register (3) of Figure 1.

In order to effectively realize such SIMD computations with the AVX instructions, we define the structures for i-particles, j-particles and the resulting forces and potentials shown in List 1. Before computing the forces on i-particles, the positions and softening lengths of i-particles are stored in the structure `Ipdata`, and the positions and masses of j-particles are in the structure `Jpdata`. The resulting forces are stored in the structure `Fodata`. Note that the structures `Ipdata` and `Fodata` contain the data of four i-particles, while the structure `Jpdata` has the data for a single j-particle.

Note that the positions, softening lengths, and forces of i-particles in the structures `Ipdata` and `Fodata` are declared as arrays of four single-precision floating-point numbers. Thus, the data on each array can be suitably loaded onto, or stored from the lower 128-bit of one YMM register. The assignment of the i-particles data shown in (1) of Figure 1 can be realized by loading the data of four i-particles onto the lower 128-bit of one YMM register, and copying the data to its upper 128-bit.

As for j-particles, since the structure `Jpdata` consists of four single-precision floating-point numbers, we can load the positions and the masses of two j-particles in one YMM-register at one time if they are aligned on the 32-byte boundaries. By broadcasting the n-th element ($n = 0, 1, 2$ and 3) in each of the lower and upper 128-bit to all the other elements, we can realize the assignment of the j-particle data as depicted in (2) of Figure 1.

After executing the gravitational force loop over j-particles, the partial forces on four i-particles exerted by different sets of j-particles are stored in the upper and lower 128-bit of a YMM register. Operating sum reduction on the upper and lower 128-bit of the YMM register, and storing the results into its lower 128-bit, we can smoothly store the results into the structure `Fodata`.

List 1. Structures for i-particles, j-particles, and the resulting forces.

```
// structure for i-particles
typedef struct ipdata{
  float x[4];
  float y[4];
  float z[4];
  float eps2[4];
} Ipdata, *pIpdata;

// structure for j-particles
typedef struct jpdata{
  float x, y, z, m;
} Jpdata, *pJpdata;

// structure for the resulting forces
// and potentials of i-particles
typedef struct fodata{
  float ax[4];
  float ay[4];
  float az[4];
  float phi[4];
} Fodata, *pFodata;
```

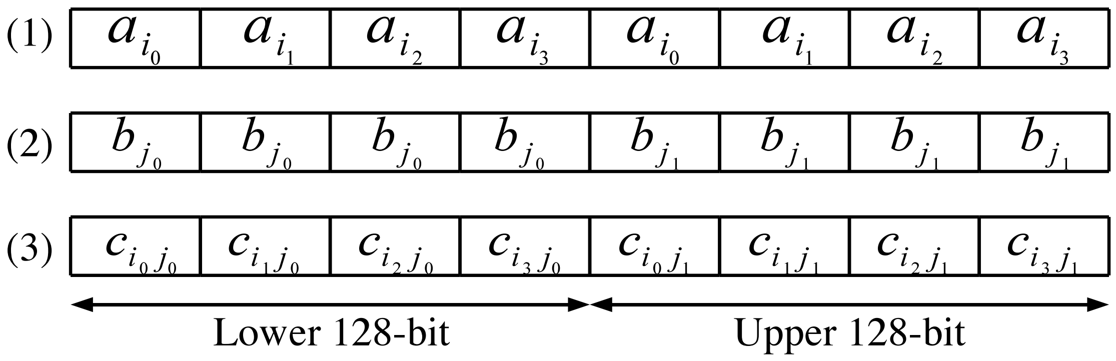


Fig. 1. Data assignments in YMM registers for SIMD calculations. The upper panel (1) shows the data assignment of four i-particles (with indices of i_0, i_1, i_2, and i_3), where the data are redundantly stored in the lower and upper 128-bit in the same order. The data of two j-particles (with indices of j_0 and j_1) are stored in the lower and upper 128-bit, respectively, as shown in the middle panel (2). The 8 values obtained by the operations between the data of four i-particles and two j-particles are stored in the order shown in the lower panel (3): $c_{ij} = f(a_i, b_j)$. For example, $c_{i_0 j_0}$ is a result of operations between a_{i_0} and b_{j_0}.

3.2. *Macros for inline assembly codes*

For the readability of the source codes shown below, let us introduce some preprocessor macros which are expanded into inline assembly codes. The definitions of the macros used in this paper are given in List 2. For macros with two and three operands, the results are stored in the second and third one, respectively, and the other operands are source operands. In these macros, operands named `src`, `src1`, `src2`, and `dst` designate the data in XMM or YMM registers, and those named `mem`, `mem64`, `mem128`, and `mem256` are data in the main memory or the cache memory, where numbers after `mem` indicate their size and alignment in bits. Brief descriptions of these macros are summarized in Table 1. More detailed explanation of the AVX instructions can be found in Intel's website[3].

List 2. Preprocessor macros for inline assembly codes.

```
#define VZEROALL asm("vzeroall");
#define VLOADPS(mem256, dst) \
  asm("vmovaps␣%0,␣%"dst::"m"(mem256));
#define VSTORPS(reg, mem256) \
  asm("vmovaps␣%"reg ",␣%0" ::"m"(mem256));
#define VLOADPS(mem128, dst) \
  asm("vmovaps␣%0,␣%"dst::"m"(mem128));
#define VSTORPS(reg, mem128) \
  asm("vmovaps␣%"reg ",␣%0" ::"m"(mem128));
#define VLOADLPS(mem64, dst) \
  asm("vmovlps␣%0,␣%"dst ",␣%"dst::"m"(mem64));
#define VLOADHPS(mem64, dst) \
  asm("vmovhps␣%0,␣%"dst ",␣%"dst::"m"(mem64));
#define VBCASTL128(src, dst) \
  asm("vperm2f128␣%0,␣%"src ",␣%"src \
  ",␣%"dst "␣"::"g"(0x00));
#define VCOPYU128TOL128(src,dst) \
  asm("vextractf128␣%0,␣%"src ",␣%"dst \
  "␣"::"g"(0x01));
#define VGATHERL128(src1,src2,dst) \
  asm("vperm2f128␣%0,␣%"src2 ",␣%"src1 \
  ",␣%"dst "␣"::"g"(0x02));
#define VCOPYALL(src,dst) \
  asm("vmovaps␣%0,␣%"src ",␣%"dst);
#define VBCAST0(src, dst) \
  asm("vshufps␣%0,␣%"src ",␣%"src \
  ",␣%"dst "␣"::"g"(0x00));
#define VBCAST1(src, dst) \
  asm("vshufps␣%0,␣%"src ",␣%"src \
  ",␣%"dst "␣"::"g"(0x55));
#define VBCAST2(src, dst) \
  asm("vshufps␣%0,␣%"src ",␣%"src \
  ",␣%"dst "␣"::"g"(0xaa));
#define VBCAST3(src, dst) \
  asm("vshufps␣%0,␣%"src ",␣%"src \
  ",␣%"dst "␣"::"g"(0xff));
#define VMIX0(src1,src2,dst) \
  asm("vshufps␣%0,␣%"src2 ",␣%"src1 \
  ",␣%"dst "␣"::"g"(0x88));
#define VMIX1(src1,src2,dst) \
  asm("vshufps␣%0,␣%"src2 ",␣%"src1 \
  ",␣%"dst "␣"::"g"(0xdd));
#define VADDPS(src1, src2, dst) \
  asm("vaddps␣" src1 "," src2 "," dst);
#define VSUBPS(src1, src2, dst) \
  asm("vsubps␣" src1 "," src2 "," dst);
#define VSUBPS_M(mem256, src, dst) \
  asm("vsubps␣%0,␣%"src ",␣%"dst \
  "␣"::"m"(mem256));
#define VMULPS(src1, src2, dst) \
  asm("vmulps␣" src1 "," src2 "," dst);
#define VRSQRTPS(src, dst) \
  asm("vrsqrtps␣" src "," dst);
#define VMINPS(src1, src2, dst) \
  asm("vminps␣"  src1 ",␣" src2 "," dst);
#define VPSRLD(imm, src1, src2) \
  asm("vpsrld␣%0,␣%"src1 ",␣%"src2::"I"(imm));
#define VPSLLD(imm, src1, src2) \
  asm("vpslld␣%0,␣%"src1 ",␣%"src2::"I"(imm));
#define PREFETCH(mem) \
  asm("prefetcht0␣%0"::"m"(mem));
```

Furthermore, we define aliases of XMM and YMM registers. Table 2 and 3 show the aliases of YMM registers in calculating Newton's force and an arbitrary shaped central force, respectively. Aliases with suffix "`_X`" indicate the lower 128-bit of the original YMM register which can be used as XMM registers for the SSE instructions. Note that some of aliases are reused for data other than described in Table 2 and 3.

3.3. *Newton's force*

Figure 3 is a schematic illustration of a force loop to compute the Newton's force on four i-particles with AVX

[3] `http://software.intel.com/en-us/avx/`

Table 1
Descriptions of the macros for inline assembly codes. One ‘value’ denotes a single-precision floating-point number.

`VZEROALL`	zero out all the YMM registers.
`VLOADPS(mem256,dst)`	load eight packed values in `mem256` to `dst`.
`VSTORPS(src,mem256)`	store eight packed values in `src` to `mem256`.
`VLOADPS(mem128,dst)`	load four packed values in `mem128` to `dst`.
`VSTORPS(src,mem128)`	store four packed values in `src` to `mem128`.
`VLOADLPS(mem64,dst)`	load two packed values in `mem64` to the lower 64-bit of the lower 128-bit in `dst`.
`VLOADHPS(mem64,dst)`	load two packed values in `mem64` to the upper 64-bit of the lower 128-bit in `dst`.
`VBCASTL128(src,dst)`	broadcast data in the lower 128-bit of `src` to the lower and upper 128-bit of `dst`.
`VCOPYU128TOL128(src,dst)`	copy the upper 128-bit in `src` to the lower 128-bit in `dst`.
`VGATHERL128(src1,src2,dst)`	copy the lower 128-bit in `src1` and `src2` to the upper 128-bit and lower 128-bit in `dst`, respectively.
`VCOPYALL(src,dst)`	copy 256-bit data from `src` to `dst`.
`VBCASTn(src,dst)`	broadcast the n-th element of each of the lower and upper 128-bit to all the other elements.
`VMIX0(src1,src2,dst)`	operate data as shown in Figure 2.
`VMIX1(src1,src2,dst)`	operate data as shown in Figure 2.
`VADDPS(src1,src2,dst)`	add `src1` to `src2`, and store the result to `dst`.
`VSUBPS(src1,src2,dst)`	subtract `src1` from `src2`, and store the result to `dst`.
`VSUBPS_M(mem256, src, dst)`	subtract `mem256` from `src`, and store the result to `dst`.
`VMULPS(src1,src2,dst)`	multiply `src1` by `src2`, and store the result to `dst`.
`VRSQRTPS(src,dst)`	compute the inverse-square-root of `src`, and store the result to `dst`.
`VMINPS(src1,src2,dst)`	compare the values in each pair of elements in `src1` and `src2`, and store the not larger ones to `dst`.
`VPSRLD(imm,src,dst)`	shift each element in the lower 128-bit of `src` left by `imm` bit, and store the result to `dst`.
`VPSRRD(imm,src,dst)`	shift each element in the lower 128-bit of `src` right by `imm` bits, and set the result to `dst`.
`PREFETCH(mem)`	prefetch data on `mem` to the cache memory.

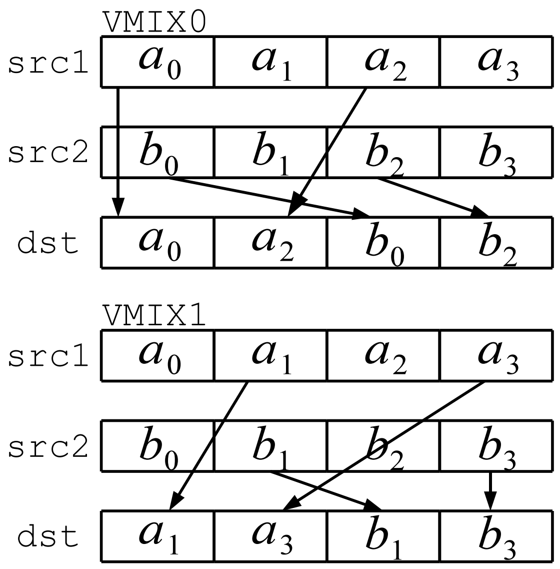
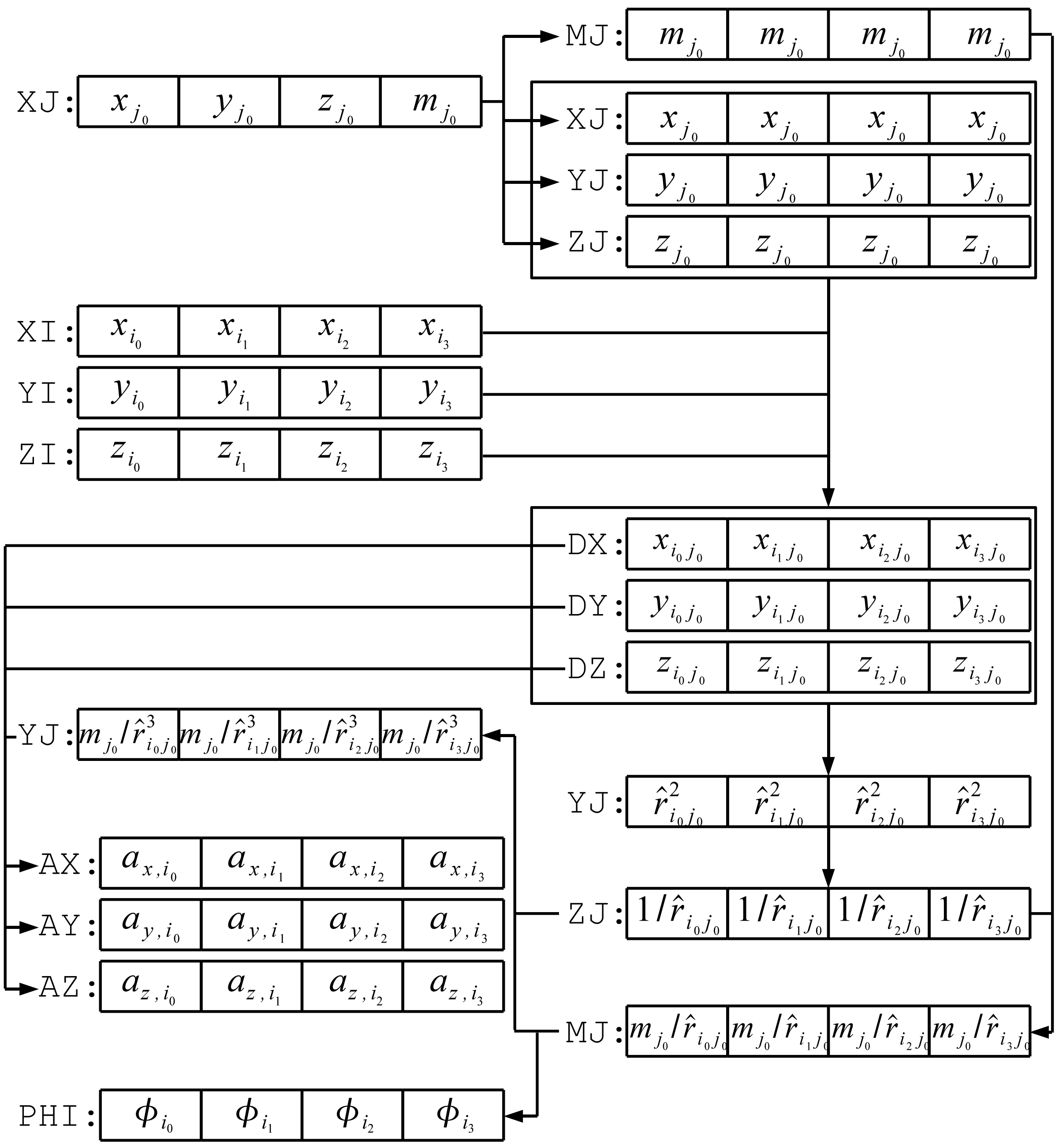


Fig. 2. Instructions `MIX0` and `MIX1`. Each set of four boxes indicates the lower (or upper) 128-bit of a YMM register. Each box contains a single-precision floating-point number.

Table 2
Aliases of YMM registers for calculating Newton’s forces in List 3.

Alias	ID	Description
`XI`	`%ymm0`	x, y, and z-coordinates of i-particles (x_i, y_i, and z_i)
`YI`	`%ymm1`	
`ZI`	`%ymm2`	
`EPS2`	`%ymm3`	square of the gravitational softening length (ϵ^2)
`AX`	`%ymm4`	forces of i-particles ($a_{x,i}$, $a_{y,i}$, and $a_{z,i}$)
`AY`	`%ymm5`	
`AZ`	`%ymm6`	
`PHI`	`%ymm7`	gravitational potentials of i-particles (ϕ_i)
`XJ`	`%ymm8`	x, y, and z-coordinates of j-particles (x_j, y_j, and z_j)
`YJ`	`%ymm9`	
`ZJ`	`%ymm10`	
`MJ`	`%ymm11`	masses of j-particles (m_j)
`DX`	`%ymm12`	relative coordinates between i- and j-particles (x_{ij}, y_{ij}, and z_{ij})
`DY`	`%ymm13`	
`DZ`	`%ymm14`	

instructions. In this figure, we depict only the lower 128-bit of YMM registers just for simplicity, while, in actual computation, the upper 128-bit is used to compute forces on the same four i-particle exerted by another j-particle.



MJ: m_{j_0} m_{j_0} m_{j_0} m_{j_0}
XJ: x_{j_0} y_{j_0} z_{j_0} m_{j_0}
XJ: x_{j_0} x_{j_0} x_{j_0} x_{j_0}
YJ: y_{j_0} y_{j_0} y_{j_0} y_{j_0}
ZJ: z_{j_0} z_{j_0} z_{j_0} z_{j_0}
XI: x_{i_0} x_{i_1} x_{i_2} x_{i_3}
YI: y_{i_0} y_{i_1} y_{i_2} y_{i_3}
ZI: z_{i_0} z_{i_1} z_{i_2} z_{i_3}
DX: $x_{i_0j_0}$ $x_{i_1j_0}$ $x_{i_2j_0}$ $x_{i_3j_0}$
DY: $y_{i_0j_0}$ $y_{i_1j_0}$ $y_{i_2j_0}$ $y_{i_3j_0}$
DZ: $z_{i_0j_0}$ $z_{i_1j_0}$ $z_{i_2j_0}$ $z_{i_3j_0}$
YJ: $m_{j_0}/\hat{r}^3_{i_0j_0}$ $m_{j_0}/\hat{r}^3_{i_1j_0}$ $m_{j_0}/\hat{r}^3_{i_2j_0}$ $m_{j_0}/\hat{r}^3_{i_3j_0}$
YJ: $\hat{r}^2_{i_0j_0}$ $\hat{r}^2_{i_1j_0}$ $\hat{r}^2_{i_2j_0}$ $\hat{r}^2_{i_3j_0}$
AX: a_{x,i_0} a_{x,i_1} a_{x,i_2} a_{x,i_3}
AY: a_{y,i_0} a_{y,i_1} a_{y,i_2} a_{y,i_3}
AZ: a_{z,i_0} a_{z,i_1} a_{z,i_2} a_{z,i_3}
ZJ: $1/\hat{r}_{i_0j_0}$ $1/\hat{r}_{i_1j_0}$ $1/\hat{r}_{i_2j_0}$ $1/\hat{r}_{i_3j_0}$
MJ: $m_{j_0}/\hat{r}_{i_0j_0}$ $m_{j_0}/\hat{r}_{i_1j_0}$ $m_{j_0}/\hat{r}_{i_2j_0}$ $m_{j_0}/\hat{r}_{i_3j_0}$
PHI: ϕ_{i_0} ϕ_{i_1} ϕ_{i_2} ϕ_{i_3}

Fig. 3. A schematic illustration of the force loop. Each set of four boxes indicates the lower 128-bit of a YMM register. Each box contains a single-precision floating-point number. Note that some of aliases are reused to store data other than described in Table 3.

The overall procedures to calculate the force on four i-particles using AVX instructions are summarized as follows:

0. Zero out all the YMM registers, and load the x, y, and z coordinates of four i-particles, and squared softening lengths to the lower 128-bit of XI, YI, ZI, and EPS2 (i.e. XI_X, YI_X, ZI_X, and EPS2_X), and copy them to the upper 128-bit of XI, YI, ZI, and EPS2, respectively.
1. Load the x, y, and z coordinates and the masses of two j-particles to XJ.
2. Broadcast the x, y, and z coordinates and the masses of two j-particles in XJ to XJ, YJ, ZJ, and MJ, respectively.
3. Subtract XI, YI, and ZI from XJ, YJ, and ZJ respectively. The results (x_{ij}, y_{ij}, and z_{ij}) are stored in DX, DY, and DZ, respectively.
4. Square x_{ij} in DX, y_{ij} in DY, and z_{ij} in DZ and sum them up to compute the squared distance between two j-particles and four i-particles. The results are stored in the alias YJ. The squared softening lengths EPS2 are also added. Eventually, the softened squared distances $\hat{r}_{ij}^2 \equiv r_{ij}^2 + \epsilon^2$ between two j-particles and four i-particles are stored in YJ.
5. Calculate inverse-square-root for $\hat{r}_{ij}^2$ in YJ, and store

Table 3
Aliases of YMM registers for calculating an arbitrary force shape in List 5.

Alias	ID	Description
X2	%ymm0	squared inter-particle distances
Y2	%ymm1	
Z2	%ymm2	
TWO	%ymm3	constant value of 2.0 in single-precision
AX	%ymm4	forces of i-particles
AY	%ymm5	
AZ	%ymm6	
R2CUT	%ymm7	cutoff radius squared
BUF0	%ymm8	buffers used to refer a look-up table
BUF1	%ymm9	
BUF2	%ymm10	
MJ	%ymm11	masses of j-particles
DX	%ymm12	relative coordinates between i- and j-particles
DY	%ymm13	
DZ	%ymm14	
ZI	%ymm15	z-components of positions of i-particles

the result $1/\hat{r}_{ij}$ in the alias ZJ.

6. Multiply $1/\hat{r}_{ij}$ in ZJ by m_j in MJ to obtain $m_j/\hat{r}_{ij}$, and store the results in MJ.
7. Accumulate $m_j/\hat{r}_{ij}$ in MJ into ϕ_i in PHI.
8. Square $1/\hat{r}_{ij}$ in ZJ, multiply the result $1/\hat{r}_{ij}^2$ by $m_j/\hat{r}_{ij}$ in MJ, and store them $m_j/\hat{r}_{ij}^3$ in YJ.
9. Multiply x_{ij} in DX, y_{ij} in DY, and z_{ij} in DZ by $m_j/\hat{r}_{ij}^3$ in YJ obtaining the forces ($m_j x_{ij}/\hat{r}_{ij}^3$, $m_j y_{ij}/\hat{r}_{ij}^3$, and $m_j z_{ij}/\hat{r}_{ij}^3$), and accumulate them into AX, AY, and AZ, respectively.
10. Return to step 1 until all the j-particles are processed.
11. Operate sum reduction of partial forces and potentials in the lower and upper 128-bits of AX, AY, AZ, and PHI, and store the results in the lower 128-bit of AX, AY, AZ, and PHI, respectively.
12. Store forces and potentials in the lower 128-bit of AX, AY, AZ, and PHI to the structure Fodata.

The function GravityKernel to compute the forces is shown in List 3. The order of instructions in List 3 is slightly different from that described above in order to obtain high issue rate of the AVX instructions by optimizing the order of operations so that operands in adjacent instruction calls do not have dependencies as much as possible. Further optimization is given by explicitly unrolling the force loop, which does not appear in the list.

List 3. A force loop to calculate the Newton's force using the AVX instructions.

```
void GravityKernel(pIpdata ipdata, pFodata fodata,
                   pJpdata jpdata, int nj)
{
  int j;

  PREFETCH(jpdata[0]);

  VZEROALL;

  VLOADPS(*ipdata->x, XI_X);
  VLOADPS(*ipdata->y, YI_X);
  VLOADPS(*ipdata->z, ZI_X);
  VLOADPS(*ipdata->eps2, EPS2_X);
  VBCASTL128(XI, XI);
  VBCASTL128(YI, YI);
  VBCASTL128(ZI, ZI);
  VBCASTL128(EPS2, EPS2);

  VLOADPS(*(jpdata), XJ);
  jpdata += 2;

  VBCAST1(XJ, YJ);
  VBCAST2(XJ, ZJ);
  VBCAST3(XJ, MJ);
  VBCAST0(XJ, XJ);

  for(j = 0 ; j < nj; j += 2) {
    VSUBPS(YI, YJ, DY);
    VSUBPS(ZI, ZJ, DZ);
    VSUBPS(XI, XJ, DX);

    VMULPS(DZ, DZ, ZJ);
    VMULPS(DX, DX, XJ);
    VMULPS(DY, DY, YJ);

    VADDPS(XJ, ZJ, ZJ);
    VADDPS(EPS2, YJ, YJ);
    VADDPS(YJ, ZJ, YJ);

    VLOADPS(*(jpdata), XJ);
    jpdata += 2;

    VRSQRTPS(YJ, ZJ);

    VMULPS(ZJ, MJ, MJ);
    VMULPS(ZJ, ZJ, YJ);

    VMULPS(MJ, YJ, YJ);
    VSUBPS(MJ, PHI, PHI);

    VMULPS(YJ, DX, DX);
    VMULPS(YJ, DY, DY);
    VMULPS(YJ, DZ, DZ);

    VBCAST1(XJ, YJ);
    VBCAST2(XJ, ZJ);
    VBCAST3(XJ, MJ);
    VBCAST0(XJ, XJ);

    VADDPS(DX, AX, AX);
    VADDPS(DY, AY, AY);
    VADDPS(DZ, AZ, AZ);
  }

  VCOPYU128TOL128(AX, DX_X);
  VADDPS(AX, DX, AX);
  VCOPYU128TOL128(AY, DY_X);
  VADDPS(AY, DY, AY);
  VCOPYU128TOL128(AZ, DZ_X);
  VADDPS(AZ, DZ, AZ);
  VCOPYU128TOL128(PHI, MJ_X);
  VADDPS(PHI, MJ, PHI);

  VSTORPS(AX_X,  *fodata->ax);
  VSTORPS(AY_X,  *fodata->ay);
  VSTORPS(AZ_X,  *fodata->az);
  VSTORPS(PHI_X, *fodata->phi);
}
```

3.4. *Central force with an arbitrary shape*

In this section, we describe how to accelerate the computation of arbitrarily shaped forces $f(r)/r$, using the AVX instructions, where $f(r)$ is a user-specified function in equation (2). Note that the inter-particle softening is also expressed in the force shape function $f(r)$, as well as the long range cut-off. Arbitrary shaped softening including the Plummer softening, $S2$ softening, etc. can be set. The function $f(r)$ is assumed to shape; almost constant at $r < \epsilon$, rapidly decreases at larger r, and reaches zero at $r = r_{\rm cut}$. Such assumptions are satisfied in the inter-particle force calculations of PPPM or TreePM methods.

In order to calculate central forces with an arbitrary shape in equation (2), we refer to a pre-calculated look-up table of $f(r)/r$ and use the linear interpolation between the sampling points. In § 3.4.1 and 3.4.2, we describe our scheme to construct the look-up table, and procedure to calculate the force by using the look-up table with the AVX instructions, respectively.

3.4.1. *Construction of an optimized look-up table*

In terms of numerical accuracy, the look-up table is preferred to have a large number of sampling points between $0 \le r \le r_{\rm cut}$. On the other hand, the size of the look-up table should be as small as possible to avoid cache misses for fast calculations. Thus, it is important how to choose sampling points of the look-up table in order to satisfy such exclusive requirements: accuracy and fast calculations of forces.

In many previous implementations, sampling points of the look-up table are chosen so that the sampling points have equal intervals in a squared inter-particle distance r^2 at $0 < r < r_{\rm cut}$. However, the sampling with equal intervals in r^2 is not a good choice, because it has coarser intervals at a smaller inter-particle distance, and the force shape at $r \lesssim \epsilon$ is poorly sampled if the number of sampling points is not large enough, while the shape at $r \simeq r_{\rm cut}$ is sampled fairly well, or even redundantly (see the top panel of Figure 6). Typically speaking, tens of thousand sampling points in the region $0 < r < r_{\rm cut}$ are required to assure the sufficient force accuracy if sampling with equal intervals in r^2 is adopted. Such look-up tables need several hundred kilobytes in single-precision, and do not fit into a low-level cache memory.

The desirable sampling of the force shapes, therefore, should have almost equal intervals in r at short distances $r \lesssim \epsilon$, and intervals proportional to r (or equal intervals in $\ln r$) at long distances. In the following, we realize such a sampling by adopting rather a new binning scheme, with which we can compute the force efficiently.

Here, we consider to construct a look-up table of $f(r)/r$ in the range of $0 < r < r_{\rm cut}$. In our binning scheme, the indices of the look-up table are calculated by directly extracting the fraction and the exponent bits of the IEEE754 format of squared inter-particle distances. First, the squared distance r^2 is affine-transformed to a single-precision floating-point number $s \equiv r^2(s_{\rm max} - 2)/r^2_{\rm cut} + 2$ so that s is in the range of $s_{\rm min} < s < s_{\rm max}$, where $s_{\rm min} \equiv 2$ and $s_{\rm max} \equiv 2^{2^E}(2 - 1/2^F)$. Here, E and F are the pre-defined positive integers, and the numbers of exponent and fraction bits extracted in computing the indices of the look-up table, respectively. Binary expressions of $s_{\rm min}$ and $s_{\rm max}$ in the IEEE754 format of single-precision (32-bit) in the case of $E = 4$ and $F = 6$ are shown in Table 4. Except that the most significant bit of the exponent part is always 1, all the bits of $s_{\rm min}$ are 0, and as for $s_{\rm max}$, only the lower E bits of the exponent and the higher F bits of the fraction are 1. Next, the indices of the look-up table for the squared distances r^2 or s are computed by extracting the lower E bits of the exponent and the higher F bits of the fraction of s (underlined portion of exponent and fraction bits in Table 4) and reinterpreting it as an integer. This procedure can be done by applying a logical right shift by $23 - F$ bits, and a bitwise-logical AND with $2^{E+F} - 1$ to s. It should be noted that the resulting size of the look-up table is 2^{E+F}.

Table 4
s-values, their exponent and fraction bits in the IEEE754 expressions, and their indices in the table for $r = 0$, $r_{\rm cut}/2$ and $r_{\rm cut}$ in the case of $E = 4$ and $F = 6$ (underlined portion of exponent and fraction bits).

r	s	exponent bits	fraction bits	index
0	2 ($s_{\rm min}$)	$1000\underline{0000}$	$\underline{000000}00000000000000000$	0
$r_{\rm cut}/2$	3.2514×10^4	$1000\underline{1101}$	$\underline{111111}00000000110000000$	895
$r_{\rm cut}$	1.3005×10^5 ($s_{\rm max}$)	$1000\underline{1111}$	$\underline{111111}00000000000000000$	1023

An affine-transformed squared distance at a sampling point with an index specified by a lower E exponent bits $b_{\rm E}$ and an upper F fraction bits $b_{\rm F}$ is expressed as

$$s_{b_{\rm E},b_{\rm F}} = 2^{b_{\rm E}+1}\left(1 + \frac{b_{\rm F}}{2^F}\right) \quad \left(0 \le b_{\rm E} < 2^E,\ 0 \le b_{\rm F} < 2^F\right). \tag{3}$$

The ratio between inter-particle distances whose affine-transformed squared distances are $s_{(b_{\rm E}+1),b_{\rm F}}$ and $s_{b_{\rm E},b_{\rm F}}$ is given by

$$\begin{aligned} \frac{r_{(b_{\rm E}+1),b_{\rm F}}}{r_{b_{\rm E},b_{\rm F}}} &= \left(\frac{s_{(b_{\rm E}+1),b_{\rm F}} - 2}{s_{b_{\rm E},b_{\rm F}} - 2}\right)^{1/2} \\ &\simeq 2^{1/2}, \end{aligned} \tag{4}$$

where $b_{\rm E} \gg 1$ is assumed for the last approximation. The interval between inter-particle distances whose affine-transformed distances are $s_{b_{\rm E},(b_{\rm F}+1)}$ and $s_{b_{\rm E},b_{\rm F}}$ is calculated as

$$\begin{aligned} r_{b_{\rm E},(b_{\rm F}+1)} - r_{b_{\rm E},b_{\rm F}} &= \left(\frac{s_{b_{\rm E},(b_{\rm F}+1)} - 2}{s_{\rm max} - 2}\right)^{1/2} - \left(\frac{s_{b_{\rm E},b_{\rm F}} - 2}{s_{\rm max} - 2}\right)^{1/2} \\ &\simeq \frac{1}{(2^F + b_{\rm F})^{1/2}}\left(\frac{2^{b_{\rm E}+1}/2^{F+2}}{s_{\rm max} - 2}\right)^{1/2}, \end{aligned} \tag{5}$$

where we also assume $b_{\rm E} \gg 1$ and $F \gg 1$ for the last approximation. Therefore, the sampling points with the same fraction bits are distributed uniformly in logarithmic scale, and those with the same exponent bits are aligned uniformly in linear scale unless the fraction bit is small.

As an example, we illustrate how the sampling points of the look-up table depend on the pre-defined integers E and F in Figure 4. We first see the cases in which either of E and F is zero, in order to see the roles of the integers E and F. As seen in Figure 4, the intervals of sampling points are roughly uniform in linear scale for the case $E = 0$ (the bottom line in the top panel), and uniform in logarithmic scale for the case $F = 0$ (the middle line in the bottom panel), unless $r/r_{\rm cut}$ is small. As expected above, the integers E and F control the number of sampling points in logarithmic and linear scales, respectively.

By comparing the sampling points with $(E, F) = (4, 0)$ and those with $(4, 2)$ (see the top panel of Figure 4), it can be seen that all intervals of the sampling points with $(E, F) = (4, 0)$ (indicated by the vertical dashed lines and double-headed arrows) are divided nearly equally into $2^F = 4$ regions by the sampling points with $(E, F) = (4, 2)$. Thus, our binning scheme is a hybrid of the linear and logarithmic binning schemes.

Figure 5 shows the comparison of the several binning in which the number of sampling points is fixed to $2^{E+F} = 2^6$. One can see that the binning with $(E, F) = (4, 2)$ has sufficient sampling points in the range of $10^{-3} \le r/r_{\rm cut} \le 10^0$, whereas the binning with the other sets of (E, F) only samples the region of $10^{-2} \le r/r_{\rm cut} < 10^0$. The number of the extracted exponent bit E should be large enough so that the scale of the softening length should be sufficiently resolved. For example, if $\epsilon/r_{\rm cut} \lesssim 10^{-2}$, E should be set to at least equal to or larger than 4.

In List 4, we present routines for constructing the look-up table. In our implementation, the look-up table contains two values: one is the force at a sampling point r_k,

$$G^0_k = \frac{f(r_k)}{r_k}, \qquad (6)$$

and the other is its difference from the next sampling point r_{k+1} divided by the interval of the affine-transformed squared distance

$$G^1_k = \frac{G^0_{k+1} - G^0_k}{s_{k+1} - s_k} \qquad (7)$$

where subscript k indicates indices of the look-up table, and is expressed as $k = 2^F \times b_{\rm E} + b_{\rm F}$. Using these two values, we can compute the linear interpolation of $f(r)/r$ at a radius r with $r_k \le r \le r_{k+1}$ by $G^0_k + (s - s_k)G^1_k$. The G^0_k and G^1_k, are stored in a two-dimensional array declared as `Force_table[TBL_SIZE][2]`, where `TBL_SIZE` is the number of the sampling points (2^{E+F}) and the values of the G^0_k and G^1_k are stored in the `Force_table[k][0]` and `Force_table[k][1]`, respectively. Since the values of G^0_k and G^1_k are stored in the adjacent memory address, we can avoid the cache misses in computing the linearly interpolated values of $f(r)/r$.

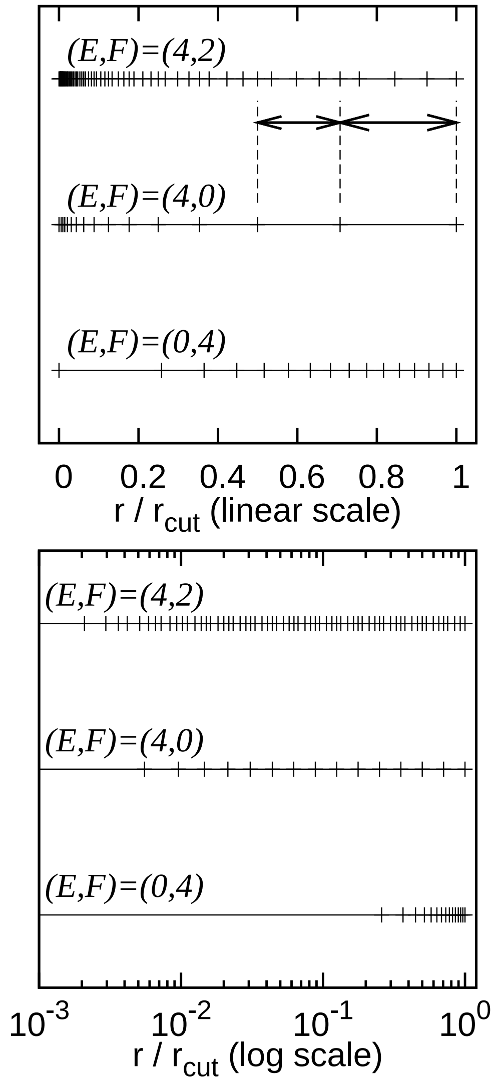


Fig. 4. Sampling points of an inter-particle distance for a look-up table in various cases of pre-defined integers E and F. The top and bottom panels take horizontal axes in linear and logarithmic scales, respectively.

List 4. Implementation of the construction of the look-up table.

```
#define EXP_BIT (4)
#define FRC_BIT (6)
#define TBL_SIZE (1 << (EXP_BIT+FRC_BIT)) // 1024

extern float Force_table[TBL_SIZE][2]; // 8 kB

union pack32{
  float f;
  unsinged int u;
};

void generate_force_table(float rcut)
{
  unsigned int tick;
  float fmax, r2scale, r2max;
  union pack32 m32;

  float force_func(float);
```

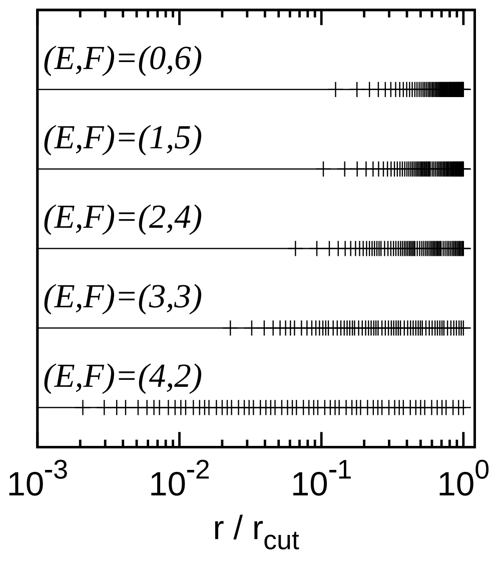


Fig. 5. Comparison of binning among the same number of sampling points in various cases of the integers E and F.

```
  tick = (1 << (23-FRC_BIT));
  fmax = (1 << (1<<EXP_BIT))*(2.0-1.0/(1<<FRC_BIT));
  r2max = rcut*rcut;
  r2scale = (fmax-2.0f)/r2max;

  for(i=0,m32.f=2.0f;i<TBL_SIZE;i++,m32.u+=tick) {
    float f, r2, r;

    f=m32.f;
    r2 = (f-2.0)/r2scale;
    float r = sqrtf(r2);
    Force_table[i][0] = force_func(r);
  }

  for(i=0,m32.f=2.0f;i<TBL_SIZE-1;i++) {
    float x0 = m32.f;
    m32.u += tick;
    float x1 = m32.f;
    float y0 = Force_table[i][0];
    float y1 = (i==TBL_SIZE-1) ? 0.0
                               : Force_table[i+1][0];
    Force_table[i][1] = (y1-y0)/(x1-x0);
  }
  Force_table[i][1] = 0.0f;
}
```

In Figure 6, we compare the conventional binning with equal intervals in squared distances to our binning with $E = 4$ and $F = 2$ (i.e. 64 sampling points), for the $S2$-force shape (Hockney & Eastwood, 1981) used in the PPPM scheme. Although we adopt $F = 5$ in the rest of this paper, we set $F = 2$ here just for good visibility of the difference of the two binning schemes. It should be noted that the number of sampling points is the same (64) in both schemes. Compared with the conventional binning scheme in the top panel, our binning scheme can faithfully reproduce the given functional form even at distances smaller than the gravitational softening length.

3.4.2. *Procedure of force calculation*

In calculating the arbitrary central forces, the data of i- and j-particles are stored in the structures `Ipdata` and

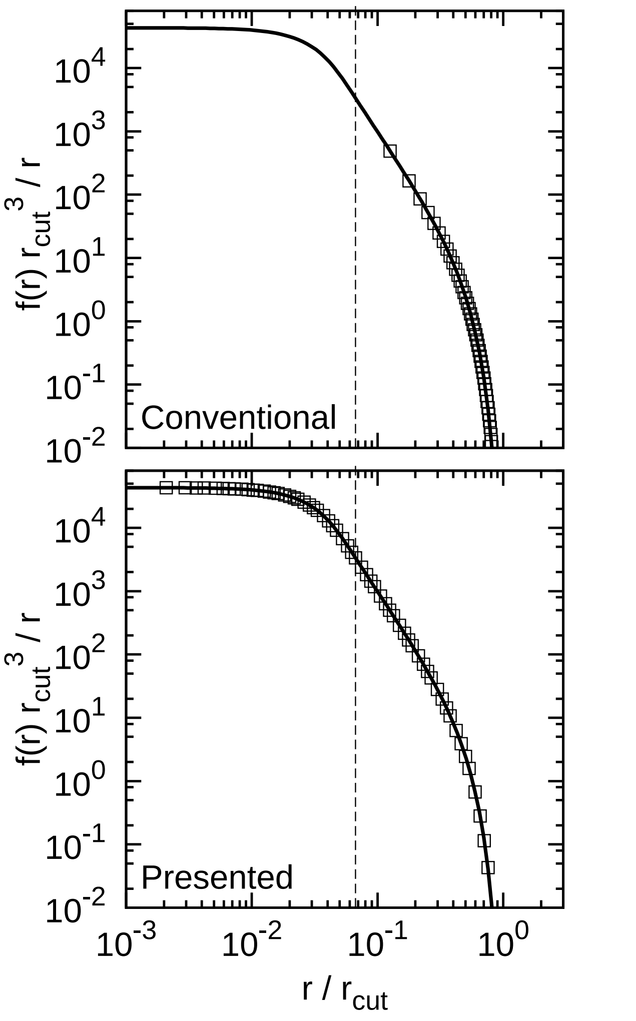


Fig. 6. Binning of $f(r)/r$ in the conventional scheme with 64 constant intervals in r^2 (top panel) and in our scheme with $E = 4$ and $F = 2$ (bottom panel) between $[0, r_{\rm cut}]$. Although we adopt $F = 5$ elsewhere in this paper, we set $F = 2$ here for viewability. $R(r, \epsilon) - R(r, r_{\rm cut})$ is assumed as a functional form of $f(r)$, in which $R(r, \eta)$ is the $S2$-profile (Hockney & Eastwood, 1981) (see equation (16)). Solid lines indicate the shape of $f(r)/r$. Vertical dashed lines in both panels are the locations of the gravitational softening length ϵ.

`Jpdata`, respectively, in the same manner as described in the case for calculating the Newton's force, except that the coordinates of i- and j-particles are scaled as

$$\tilde{\boldsymbol{r}}_i = \frac{\boldsymbol{r}_i}{r_{\rm cut}/\sqrt{s_{\rm max} - 2}}, \tag{8}$$

so that we can quickly compute the affine-transformed squared inter-particle distances between i- and j-particles. As in the case of the Newton's force, we compute the forces of four i-particles exerted by two j-particles using the AVX instructions. Using the scaled positions of the particles, the calculation of the forces is performed in the force loop as follows;

(i) Calculate an affine-transformed distance between i- and j-particles, s, as

$$s = \min\left(|\tilde{\boldsymbol{r}}_j - \tilde{\boldsymbol{r}}_i|^2 + 2, s_{\rm max}\right), \tag{9}$$

where the function "min" returns the minimum value among arguments.

(ii) Derive an index k of the look-up table from the affine-transformed squared distance, s, computed in the previous step by applying a bitwise-logical right shift by $23-F$ bits and reinterpreting the result as an integer.

(iii) Refer to the look-up table to obtain G_k^0 and G_k^1. Note that the address of the pointer to `fcut` is decremented by `1<<(30-(23-F))` in advance (see line 24 in List 5) to correct the effect of the most significant exponential bit of s.

(iv) Derive an affine-transformed distance s_k that corresponds to the k-th sampling point r_k by applying a bitwise-logical left shift by $23-F$ bits to k and reinterpreting the result as a single-precision floating-point number.

(v) Compute the value of $f(|\boldsymbol{r}_j - \boldsymbol{r}_i|)/|\boldsymbol{r}_j - \boldsymbol{r}_i|$ by the linear interpolation of G_k^0 and G_{k+1}^0. Using the values of G_k^0 and G_k^1, the interpolation can be performed as

$$\frac{f(|\boldsymbol{r}_j - \boldsymbol{r}_i|)}{|\boldsymbol{r}_j - \boldsymbol{r}_i|} = G_k^0 + G_k^1\,(s - s_k)\,. \qquad (10)$$

(vi) Accumulate scaled "forces" on i-particles as

$$\tilde{\boldsymbol{a}}_i = \sum_j^N m_j \frac{f(|\boldsymbol{r}_j - \boldsymbol{r}_i|)}{|\boldsymbol{r}_j - \boldsymbol{r}_i|}(\tilde{\boldsymbol{r}}_j - \tilde{\boldsymbol{r}}_i) \qquad (11)$$

After the force loop, the scaled "forces" are rescaled back as

$$\boldsymbol{a}_i = \frac{r_{\mathrm{cut}}}{\sqrt{s_{\max} - 2}}\tilde{\boldsymbol{a}}_i. \qquad (12)$$

The actual code of the force loop for the calculation of the central force with an arbitrary force shape is shown in List 5. Note that bitwise-logical shift instructions such as `VPSRLD` and `VPSLLD` can be operated only to XMM registers or the lower 128-bit of YMM registers. In order to operate bitwise-logical shift instructions to data in the upper 128-bit of a YMM register, we have to copy the data to the lower 128-bit of another YMM register. Bitwise-logical shift operations to the upper 128-bit of YMM registers are supposed to be implemented in the future AVX2 instruction set. Also note that we cannot refer to the look-up table in a SIMD manner and have to issue the `VLOADLPS` and `VLOADHPS` instructions one by one (see lines 89–92 and 94–97 in List 5). Except for those operations, all the other calculations are performed in a SIMD manner using the AVX instructions.

List 5. Implementation of arbitrary force calculation using AVX instructions.

```
 1  #define FRC_BIT (6)
 2  #define ALIGN32 __attribute__ ((aligned(32)))
 3  #define ALIGN64 __attribute__ ((aligned(64)))
 4
 5  typedef float  v4sf __attribute__ ((vector_size(16)));
 6  typedef struct ipdata_reg{
 7    float x[8];
 8    float y[8];
 9  } Ipdata_reg, *pIpdata_reg;
10
11  void GravityKernel(pIpdata ipdata,
12                     pJpdata jp,
13                     pFodata fodata,
14                     int nj,
15                     float fcut[][2],
16                     v4sf r2cut, v4sf accscale)
17  {
18    int j;
19    unsigned long int ALIGN64 idx[8]
20    = {0, 0, 0, 0, 0, 0, 0, 0};
21    Ipdata_reg ALIGN32 ipdata_reg;
22    static v4sf two = {2.0f, 2.0f, 2.0f, 2.0f};
23
24    fcut -= (1<<(30-(23-FRC_BIT)));
25
26    VZEROALL;
27
28    VLOADPS(ipdata->x[0], X2_X);
29    VLOADPS(ipdata->y[0], Y2_X);
30    VLOADPS(ipdata->z[0], Z2_X);
31    VLOADPS(r2cut, R2CUT_X);
32    VLOADPS(two, TWO_X);
33    VBCASTL128(X2, X2);
34    VSTORPS(X2, ipdata_reg.x[0]);
35    VBCASTL128(Y2, Y2);
36    VSTORPS(Y2, ipdata_reg.y[0]);
37    VBCASTL128(Z2, ZI);
38    VBCASTL128(R2CUT, R2CUT);
39    VBCASTL128(TWO, TWO);
40
41    VLOADPS(*jp, MJ);
42    jp += 2;
43
44    VBCAST0(MJ, X2);
45    VBCAST1(MJ, Y2);
46    VBCAST2(MJ, Z2);
47
48    VSUBPS_M(*ipdata_reg.x, X2, DX);
49    VMULPS(DX, DX, X2);
50    VADDPS(TWO, X2, X2);
51
52    VSUBPS_M(*ipdata_reg.y, Y2, DY);
53    VMULPS(DY, DY, Y2);
54    VADDPS(X2, Y2, Y2);
55
56    VSUBPS(ZI, Z2, DZ);
57    VMULPS(DZ, DZ, Z2);
58    VADDPS(Y2, Z2, Y2);
59
60    VBCAST3(MJ, MJ);
61    VMULPS(MJ, DX, DX);
62    VMULPS(MJ, DY, DY);
63    VMULPS(MJ, DZ, DZ);
64
65    VMINPS(R2CUT, Y2, Z2);
66
67    for(j = 0; j < nj; j += 2){
68      VLOADPS(*jp, MJ);
69      jp += 2;
70
71      VCOPYU128TOL128(Z2, Y2_X);
72      VPSRLD(23-FRC_BIT, Y2_X, Y2_X);
73      VPSRLD(23-FRC_BIT, Z2_X, X2_X);
74
75      VSTORPS(X2_X, idx[0]);
76      VSTORPS(Y2_X, idx[4]);
77
78      VPSLLD(23-FRC_BIT, Y2_X, Y2_X);
79      VPSLLD(23-FRC_BIT, X2_X, X2_X);
80
81      VGATHERL128(Y2, X2, Y2);
82      VSUBPS(Y2, Z2, Z2);
83
84      VBCAST0(MJ, X2);
85      VBCAST1(MJ, Y2);
86      VSUBPS_M(*ipdata_reg.x, X2, X2);
87      VSUBPS_M(*ipdata_reg.y, Y2, X2);
88
89      VLOADLPS(*fcut[idx[4]], BUF0_X);
90      VLOADHPS(*fcut[idx[5]], BUF0_X);
91      VLOADLPS(*fcut[idx[0]], BUF1_X);
92      VLOADHPS(*fcut[idx[1]], BUF1_X);
```

```
      VGATHERL128(BUF0, BUF1, BUF1);
      VLOADLPS(*fcut[idx[6]], BUF2_X);
      VLOADHPS(*fcut[idx[7]], BUF2_X);
      VLOADLPS(*fcut[idx[2]], BUF0_X);
      VLOADHPS(*fcut[idx[3]], BUF0_X);
      VGATHERL128(BUF2, BUF0, BUF2);
      VMIX1(BUF1, BUF2, BUF0);
      VMIX0(BUF1, BUF2, BUF2);

      VMULPS(Z2, BUF0, BUF0);

      VBCAST2(MJ, Z2);
      VBCAST3(MJ, MJ);
      VSUBPS(ZI, Z2, Z2);

      VADDPS(BUF0, BUF2, BUF2);
      VMULPS(BUF2, DX, DX);
      VMULPS(BUF2, DY, DY);
      VMULPS(BUF2, DZ, DZ);

      VADDPS(DX, AX, AX);
      VADDPS(DY, AY, AY);
      VADDPS(DZ, AZ, AZ);

      VCOPYALL(X2, DX);
      VCOPYALL(Y2, DY);
      VCOPYALL(Z2, DZ);

      VMULPS(X2, X2, X2);
      VMULPS(Y2, Y2, Y2);
      VMULPS(Z2, Z2, Z2);

      VADDPS(TWO, X2, X2);
      VADDPS(Z2, Y2, Y2);
      VADDPS(X2, Y2, Y2);

      VMULPS(MJ, DX, DX);
      VMULPS(MJ, DY, DY);
      VMULPS(MJ, DZ, DZ);
      VMINPS(R2CUT, Y2, Z2);
    }
    VCOPYU128TOL128(AX, X2_X);
    VADDPS(AX, X2, AX);
    VCOPYU128TOL128(AY, Y2_X);
    VADDPS(AY, Y2, AY);
    VCOPYU128TOL128(AZ, Z2_X);
    VADDPS(AZ, Z2, AZ);

    VMULPS_M(accscale, AX_X, AX_X);
    VMULPS_M(accscale, AY_X, AY_X);
    VMULPS_M(accscale, AZ_X, AZ_X);

    VSTORPS(AX_X, *fodata->ax);
    VSTORPS(AY_X, *fodata->ay);
    VSTORPS(AZ_X, *fodata->az);
}
```

Although the AVX instruction set takes the non-destructive 3-operand form, the copy instruction between registers appeared in the code above, which was intended to avoid the inter-register dependencies.

3.5. *Parallelization on multi-core processors*

On multi-core processors, we can parallelize the calculations of the forces of i-particles for both of the Newton's force and arbitrary central forces using the OpenMP programming interface by assigning a different set of four i-particles onto each processor core. List 6 shows a code fragment for the parallelization of the computations of the Newton's force. The calculation of an arbitrary force can be parallelized similarly to that of Newton's force.

List 6. Code fragment to parallelize the calculations using OpenMP programming interface.

```
#define ISIMD 4

extern Ipdata ipos[NI_MEMMAX / ISIMD];
extern Jpdata jpos[NJ_MEMMAX];
extern Fodata iacc[NI_MEMMAX / ISIMD];

int nig = ni / ISIMD + (ni % ISIMD ? 1 : 0)

#pragma omp parallel for
for(i = 0; i < nig; i++)
  GravityKernel(&ipos[i], &iacc[i], jpos, nj);
```

3.6. *Application programming interfaces*

With the implementations of the force calculation accelerated with the AVX instructions described above, we develop a set of application programming interfaces (APIs) for N-body simulations, which is compatible to GRAPE-5 library[4], except that our library do not support functions to search for neighbours of a given particle. The APIs are shown in List 7. g5_set_xmj sends the data of j-particles to the array of the structure Jpdata. g5_calculate_force_on_x sends the data of i-particles to the array of the structure Ipdata, and computes the forces and potentials of i-particles and returns them into the arrays ai and pi, respectively.

In the function g5_open, we derive statistical bias of the fast approximation of inverse-square-root, VRSQRTPS instruction. As Nitadori et al. (2006) reported, the results of this instruction contains a bias which is implementation-dependent. We statistically correct this bias in the same way as Nitadori et al. (2006).

Softening length and the number of j-particles are set by the functions g5_set_eps_to_all and g5_set_n, respectively. g5_close does nothing and is just for compatibility with the GRAPE-5 library.

List 8 shows a code fragment to perform an N-body simulation, using this APIs.

List 7. APIs.

```
void g5_open(void);
void g5_close(void);
void g5_set_eps_to_all(double eps);
void g5_set_n(int nj);
void g5_set_xmj(int adr,
                int nj,
                double (*xj)[3],
                double *mj);
void g5_calculate_force_on_x(double (*xi)[3],
                             double (*ai)[3],
                             double *pi,
                             int ni);
```

List 8. Sample code.

```
int n;              // The number of particles
double m[NMAX];     // Mass
double x[NMAX][3]; // Position
double v[NMAX][3]; // Velocity
double a[NMAX][3]; // Force
```

[4] http://www.kfcr.jp/downloads/g7pkg2.2.1/g5user.pdf

```
double p[NMAX];     // Potential
double t;           // Time
double tend;        // Time at the finish time
double dt;          // Timestep
void time_integrator(int,
                          double (*)[3],
                          double (*)[3],
                          double (*)[3]
                          double);
                        // Function for time integration

g5_open();
g5_set_eps_to_all(eps);
g5_set_n(n);
while(t < tend){
  g5_set_xmj(0,n,x,m);
  g5_calculate_force_on_x(x,a,p,n);
  time_integrator(n,x,v,a,dt);
  t += dt;
}
g5_close();
```

For the version of arbitrary force shape, we provide a new API call to set the force-table through a function pointer, which is not compatible to the GRAPE-5 API.

4. Accuracy

4.1. *Newton's force*

We investigate accuracy of forces and potentials obtained by our implementation for Newton's force. For this purpose, we compute the forces and potentials of particles in the Plummer models using our implementations and compare them with those computed fully in double-precision floating-point numbers without any explicit use of the AVX instructions. For the calculations of the forces and the potentials, we adopt the direct particle-particle method and the softening length of $4r_{\rm v}/N$, where $r_{\rm v}$ is a virial radius of the Plummer model and N is the number of particles.

Figure 7 shows the cumulative distribution of relative errors in the forces and the potentials of particles,

$$\frac{|\boldsymbol{a}_{\rm AVX} - \boldsymbol{a}_{\rm DP}|}{|\boldsymbol{a}_{\rm DP}|}, \tag{13}$$

and

$$\frac{|\phi_{\rm AVX} - \phi_{\rm DP}|}{|\phi_{\rm DP}|}, \tag{14}$$

where $\boldsymbol{a}_{\rm AVX}$ and $\phi_{\rm AVX}$ are the force and the potential calculated using our implementation, and $\boldsymbol{a}_{\rm DP}$ and $\phi_{\rm DP}$ are those computed fully in double-precision. It can be seen that most of the particles have errors less than 10^{-4}. These errors primarily come from the approximate inverse-square-root instruction `VRSQRTPS`, whose accuracy is about 12-bit, and consistent with the typical errors of $\simeq 10^{-4}$.

While the errors of the forces are distributed down to less than 10^{-7}, the errors of the potentials are mostly larger than $\simeq 3 \times 10^{-5}$. It can be ascribed to the way of excluding the contribution of self-interaction to the potentials. In computing a potential of the i-th particle, we accumulate the contribution from particle pairs between the i-th particle and all the particles including itself, and then subtract the contribution of the potential between the i-th particle and itself, $-m_i/\epsilon$ to finally obtain the correct potential of the i-th particle. Note that the potential between the i-th particle and itself is largest among the potentials between the i-th particle and all the particles, since the separation between i-particle and itself is zero. Thus, the subtraction of the "potential" due to the self-interaction causes the cancellation of the significant digits, and consequently degrades the accuracy of the potentials.

A remedy for such degradation of the accuracy is to avoid the self-interaction in the force loop. In fact, we do so in calculating the potentials in double-precision ($\phi_{\rm DP}$) in Figure 7. However, such treatment requires conditional bifurcation inside the force loop, and significantly reduces the computational performance. The potentials of particles are usually necessary only for checking the total energy conservation, and the accuracy obtained in our implementation is sufficient for that purpose. For these reasons, we choose the original way to compute the potentials of particles in our implementation.

4.2. *Central force with an arbitrary shape*

In order to see accuracies of central forces with an arbitrary shape obtained in our implementation, we choose a force shape which is frequently adopted in cosmological N-body simulations using PPPM or TreePM methods. Such methods are comprised of the particle–mesh (PM) and the particle–particle (PP) parts which compute long- and short-range components of inter-particle forces, respectively. Our implementation of the calculation of arbitrarily-shaped central forces can accelerate the calculation of the PP part, in which the force shape is different from the Newton's force and is expressed as

$$f(r) = R(r, \epsilon) - R(r, r_{\rm cut}), \tag{15}$$

where $R(r, a)$ is the so-called $S2$-profile with a softening length of a (Hockney & Eastwood, 1981) given by

$$R(r,a) = \begin{cases} \left(224\xi - 224\xi^3 + 70\xi^4 + 48\xi^5 - 21\xi^6\right)/35a^2 \\ \quad \text{for } (0 \le \xi < 1) \\ \left(12/\xi^2 - 224 + 896\xi - 840\xi^2 + 224\xi^3 + 70\xi^4\right. \\ \quad \left. -48\xi^5 + 7\xi^6\right)/35a^3 \text{ for } (1 \le \xi < 2) \\ \dfrac{1}{r^2} \text{ for } (2 \le \xi) \end{cases}. \tag{16}$$

We calculate forces exerted between 4K particle pairs with various separations uniformly distributed in $\ln(r)$ in the range of $5 \times 10^{-3} < r/r_{\rm cut} < 1$ using our implementation described in section 3.4, where 1K is equal to 1024. We set ϵ and $r_{\rm cut}$ to 3.125×10^{-3} and 4.6875×10^{-2}, and masses to unity. In creating the look-up table of the force shape, we set $E = 4$ and $F = 5$.

Figure 8 shows a functional form of $R(r, \epsilon)$ (solid curve) and $f(r)$ (dashed curve) in the top panel and relative errors of forces including both PP and PM parts, i.e. $R(r, \epsilon)$, in

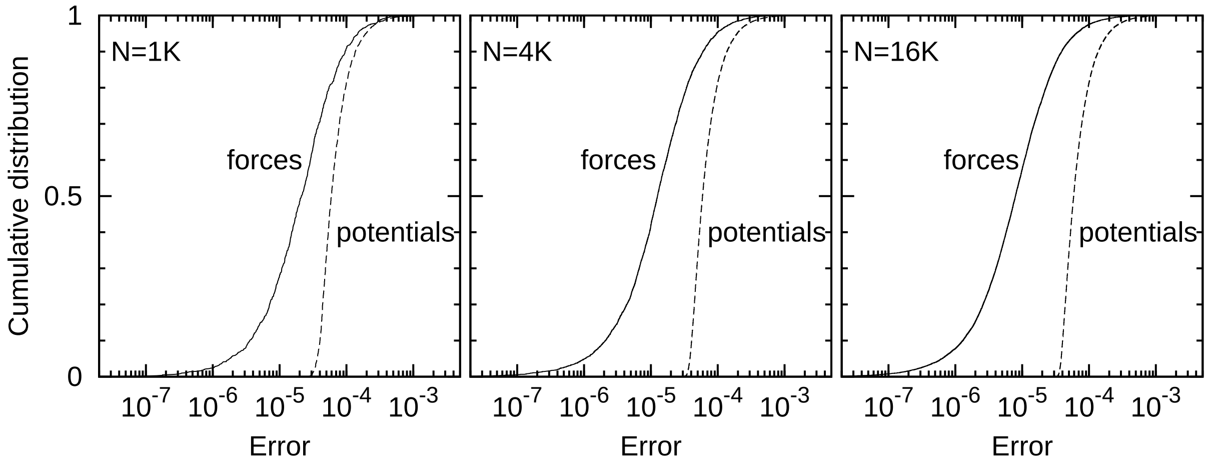


Fig. 7. Cumulative distribution of errors in forces (solid curves) and potentials (dashed curves) of particles in Plummer models with $N = 1\mathrm{K}$, 4K, and 16K, where 1K is equal to 1024. Softening lengths are set to $4r_\mathrm{v}/N$. The errors are calculated as $|\boldsymbol{a}_\mathrm{AVX} - \boldsymbol{a}_\mathrm{DP}|/|\boldsymbol{a}_\mathrm{DP}|$ and $|\phi_\mathrm{AVX} - \phi_\mathrm{DP}|/|\phi_\mathrm{DP}|$, where $\boldsymbol{a}$ is force, ϕ potential. The subscripts of "AVX" and "DP" indicate the forces and potentials obtained as our implementation in section 3.3, and those obtained by performing all the calculations in double-precision, respectively. We deal self-interactions as described in the text.

the bottom panel as a function of r/r_cut. In Figure 8, we can see that the relative errors are less than 10^{-3}, which are sufficiently accurate for cosmological N-body simulations.

5. Performance

In this section, we present the performance of our implementation of the collisionless N-body simulation using the AVX instructions (hereafter AVX-accelerated implementation). For the measurement of the performance, we use an Intel Core i7–2600 processor with 8MB cache memory and a frequency of 3.40 GHz, which contains four processor cores. In measuring the performance, Intel Turbo Boost Technology is disabled, and Intel Hyper-Threading Technology (HTT) is enabled. A compiler which we adopt is `GCC 4.4.5`, with options `-O3 -ffast-math -funroll-loops`. To see the advantage of the AVX instructions relative to the SSE instructions, we also develop the implementations with the SSE instructions rather than the AVX instructions both for Newton's force and arbitrarily-shaped force (SSE-accelerated implementation).

5.1. *Newton's force*

First, we show the performance of our implementation for Newton's force. The performance is measured by executing the direct particle-particle calculation of the Plummer model with the number of particles from 0.5K to 32K. The left panel of Figure 9 depicts the performances of the AVX- and SSE-accelerated implementations. For comparison, we also show the performance of an implementation without any explicit use of SIMD instructions (labeled as "w/o SIMD" in the left panel of Figure 9). The numbers of interactions per second are 2×10^9 in the case of the AVX-accelerated implementation with a single thread, which corresponds to 75 GFLOPS, where the number of floating-point operations for the computation of force and potential for one pair of particles is counted to be 38. The performances of the SSE- and AVX-accelerated implementations with a single thread are higher than those without SIMD instructions by 10 and 20 times, respectively, and higher than those expected from the degree of concurrency of the SSE and AVX instructions for single-precision floating-point number (4 and 8, respectively). This is because a very fast instruction of approximate inverse-square-root is not used in the "w/o SIMD" implementation. On the other hand, the performance with the AVX-accelerated implementation is higher than that of the SSE-accelerated implementation roughly by a factor of two as expected.

Furthermore, in the left panel of Figure 9, we show the performance of a GPU-accelerated N-body code based on the direct particle-particle method implemented using the CUDA language, where the GPU board is NVIDIA GeForce GTX 580 connected through the PCI-Express Gen2 x16 link. The GPU-accelerated N-body code computes the forces and potentials of the particles using GPUs, and integrate the equations of the motion of the particles on a CPU. Thus, the communication of the particle data between the main memory of the host machine and the device memory on the GPU boards is required, and can hamper the total efficiency of the code. Of course, if all the calculations are performed on GPUs, we might not suffer from such overhead. However, the performance of such implementation cannot be fairly compared with those of the AVX- and SSE-accelerated implementations, because the communication of the particle data is inevitable when we perform N-body simulations with multiple GPUs or with multiple nodes equipped with GPUs, regardless of the N-body solvers such as Tree and TreePM methods.

The performances of the AVX- and SSE-accelerated implementations are almost independent of the total number

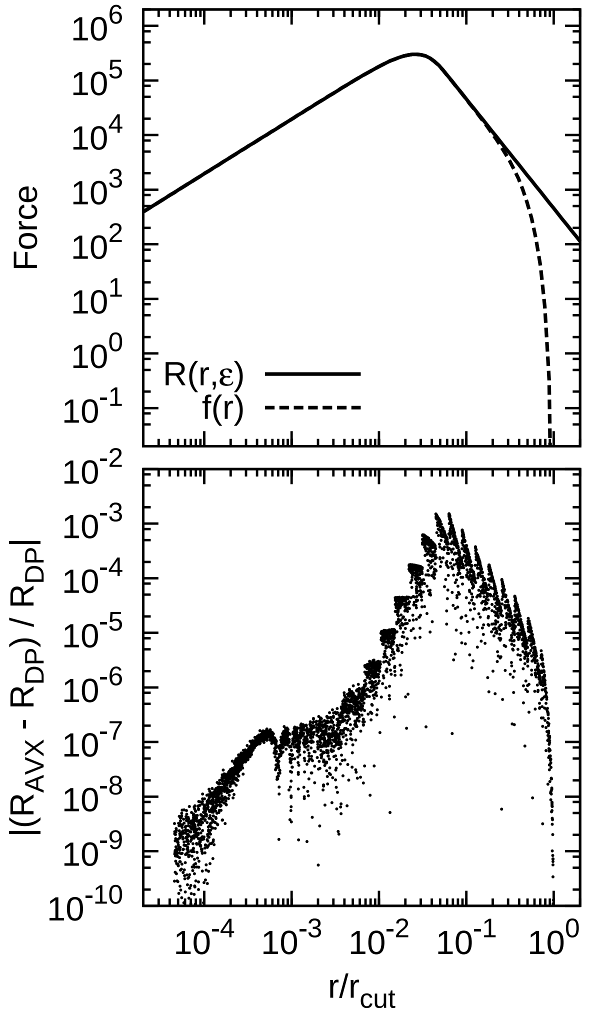


Fig. 8. Shape of $R(r, \epsilon)$ and $f(r)$ (upper panel) and the relative errors of forces of particle pairs with a separation r (bottom panel) as a function of $r/r_{\rm cut}$, where the forces include both PP and PM parts. Here, $R_{\rm AVX}$ and $R_{\rm DP}$ are, respectively, an absolute force calculated with our implementation and that obtained by performing all the calculations in double-precision without referring to the look-up table. The separations of particle pairs are distributed uniformly in $\ln(r)$ in the range of $5 \times 10^{-3} < r/r_{\rm cut} < 1$.

of particles, N. On the other hand, the performance of the GPU-accelerated implementation strongly depends on the number of particles N, due to the non-negligible overhead caused by the particle data communication. For $N = 0.5$K, the performance of the GPU-accelerated implementation is only 5% of that for $N = 32$K. Thus, for small N (0.5K and 1K), the performance of the AVX-accelerated implementation with four threads is higher than that with GPU-accelerated implementation, although, for large N (4K–32K), the performance of the GPU-accelerated implementation is higher than that of the AVX-accelerated implementation. These features can be explained by the communication overhead in the GPU-accelerated implementation.

So far, we see the performance of our code in the case that both the numbers of i- and j-particles (N_i and N_j, respectively) are the same and equal to N. However, in actual computations of forces in collisionless N-body simulations based on various N-body solvers such as PPPM, Tree, and TreePM methods, the numbers of i- and j-particles N_i and N_j are much smaller than the total number of particles N. In the Tree method modified for the effective force with external hardwares or softwares as described in Makino (1991), for example, N_i is the number of particles, for which a tree traverse is performed simultaneously and the resultant interaction list (size N_j) is shared, and typically around 10–1000. Furthermore, if one adopts the individual timestep algorithm, the number of i-particles N_i gets even smaller. The number of j-particles N_j is also decreased in Tree and TreePM methods. Therefore, we show the performance for typical N_i and N_j in the realistic situations of typical collisionless N-body simulations.

The right panel of Figure 9 shows the performance of the AVX-accelerated implementation using four threads with four processor cores (black lines) and that of the GPU-accelerated one (red lines) for various set of N_i and N_j. It can be seen that the obtained performance gets lower for the smaller N_i and N_j, regardless of the implementations. For the AVX- and SSE-accelerated implementations, this feature is due to the overhead of storing the particle data into the structures `Ipdata` and `Jpdata` shown in List 1. The amount of the overhead of storing i- and j-particles are proportional to N_i and N_j, respectively, and the computational cost is proportional to $N_i N_j$. Keeping this in mind the low performance with $N_i = 16$ compared with those with $N_i \geq 64$ can be ascribed to the overhead of storing j-particles to the structure `Jpdata`. For the GPU-accelerated implementation, the overhead originates from the transfer of the particle data to the memory on GPUs. It can be seen that the performance of the AVX-accelerated implementation has rather mild dependence on N_i and N_j, while that of the GPU-accelerated one relatively strongly depends on N_j. Such difference reflects the fact that the bandwidths and latency of the communication between GPUs and CPUs are rather poor compared with those of memory access between CPUs and main memory. Thus, the performance of the GPU-accelerated implementation is apparently superior to the AVX-accelerated one only when both of N_i and N_j are sufficiently large (say, $N_i > 1$K and $N_j > 4$K).

At the end of this section, we apply our AVX-accelerated implementation to Barnes-Hut Tree method (Barnes & Hut, 1986), and measure its performance. Our tree code is based on the PP part of TreePM code implemented by Yoshikawa & Fukushige (2005) and Fukushige et al. (2005), in which they accelerated the calculations of the gravitational forces of the $S2$-profile using GRAPE-5 and GRAPE-6A systems under the periodic boundary condition. We modify the tree code such that it can compute the Newton's force under the vacuum boundary condition. Since both of GRAPE-6A systems and Phantom-GRAPE library support the same APIs, we can easily utilize the capability of Phantom-GRAPE by simply exchanging the software library.

Using the tree code described above, we calculate gravitational forces and potentials of all the particles in a Plum-

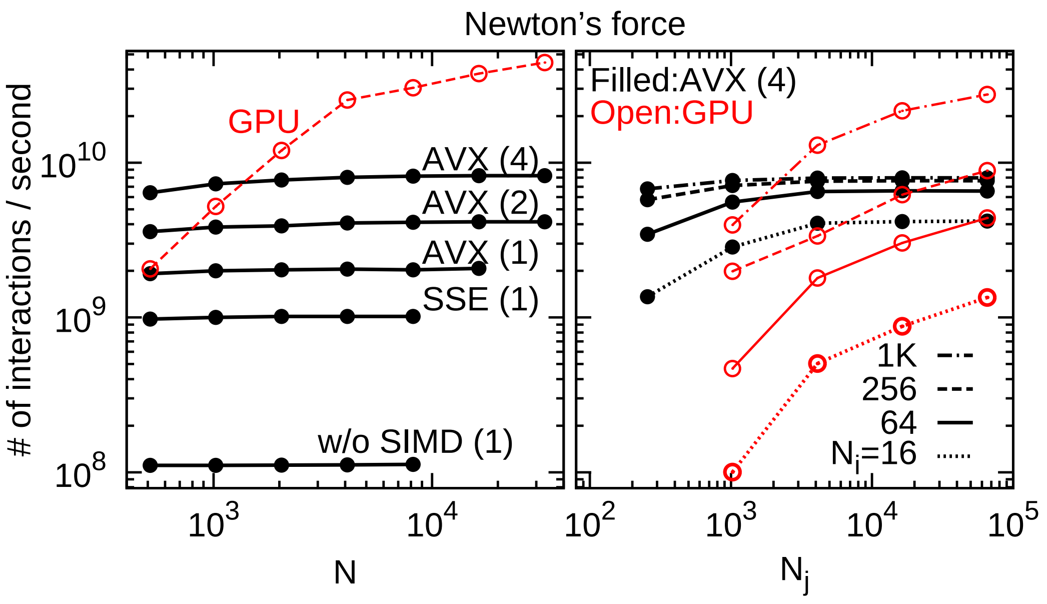


Fig. 9. (Left) Performance of the AVX- and SSE-accelerated implementation for Newton's force as a function of the number of particles (solid lines). The numbers in parentheses refer to the number of threads adopted. Dashed curve shows the performance of a GPU (NVIDIA GeForce GTX 580) attached to a host computer equipped with an Intel Core i7–2600K processor. (Right) Performance of the calculations of forces and potentials for a various set of the number of i- and j-particles, N_i and N_j, respectively. Black and red curves show the performance with the AVX-accelerated implementation on four threads and that of GPU-accelerated implementation.

mer model and a King model with the dimensionless central potential depth $W_0 = 9$. We measure the performance on an Intel Core i7–2600 processor. For the comparison with other codes, we also measure the performance of the same code but without any explicit use of SIMD instructions, and the publicly available code `bonsai` (Bédorf et al., 2012), which is a GPU-accelerated N-body code based on the tree method. The performance of the `bonsai` code is measured on a system with NVIDIA GeForce GTX 580. Since the `bonsai` code utilizes the quadrupole moments of the particle distribution in each tree node as well as the monopole moments in the force calculations, for a fair comparison of the performance with the `bonsai` code, we give our tree code a capability to use the quadrupole moments in each tree node, although the original code uses only the monopole moments. We represent these multipole moments as pseudo-particles, using pseudo-particle multipole method (Kawai & Makino, 2001). Figure 10 shows the wall clock time to compute gravitational forces and potentials for each tree code. We show the both results with the code which uses the quadrupole moments (lower panels) and the one which uses only the monopole moments (upper panels). Note that the wall clock time includes the time for tree construction, tree traverse and calculations of forces and potentials but we exclude the time to integrate orbital motion of particles. As expected, the wall clock time with the AVX-accelerated implementation is roughly 10 times shorter than those without any explicit use of SIMD instructions, owing to parallelism to calculate forces and potentials. The wall clock time with the AVX-accelerated implementation is about only three times longer than those with `bonsai`, despite that theoretical peak performance of Intel Core i7–2600 (220 GFLOPS) is lower than that of NVIDIA GeForce GTX 580 (1600 GFLOPS) by a factor of 7.3 in single-precision. We expect that the performance of our AVX-accelerated implementation could be close to that of the `bonsai` in the following situations. When we adopt individual timestep algorithm, the number of i-particles is effectively decreased, and a part of GPU cores becomes inactive. Thus, the performance of GPU-accelerated implementation would be degraded more rapidly than that of our AVX-accelerated implementation. Furthermore, when we use GPU-accelerated implementation on massively parallel environments, the communication between CPUs and GPUs is inevitable, which also degrades the performance of GPU-accelerated implementation.

5.2. *Force with an arbitrary shape*

The left panel of Figure 11 shows the performance of our implementation to calculate forces with an arbitrary force shape accelerated with the AVX and SSE instructions. For the comparison, we also plot the performance of an implementation without any explicit use of the SIMD instructions. The numbers of exponent and fraction bits used to referring the look-up table are set to $E = 4$ and $F = 6$, respectively. The performance of the AVX-accelerated implementation with a single thread is 2 and 6 times higher than that of the SSE-accelerated one and the one without any SIMD instructions, respectively. These forces with the use of the AVX instructions are lower than those expected from the degree of concurrency of their SIMD operations, 8, mainly because the reference of a look-up table is not car-

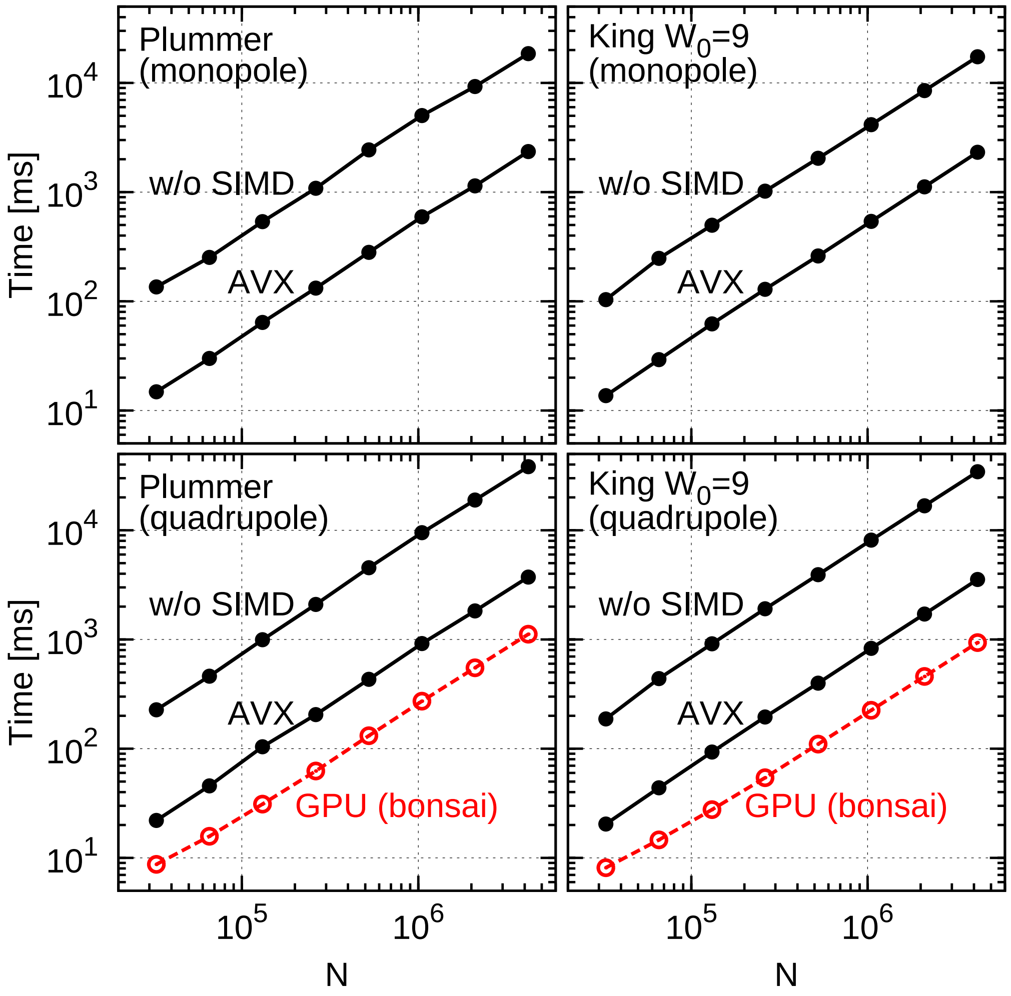


Fig. 10. Wall clock Time for calculating forces and potentials of all the particles in the Plummer model (left panels) and the King model (right panels) using Barnes-Hut tree method. The performance for codes which utilize only the monopole moment of the particle distribution in each tree nodes (upper panels) and ones which utilize monopole and quadrupole moments (lower panels) are shown. The opening parameter is set to $\theta = 0.75$. The performances of "w/o SIMD" and AVX-accelerated implementation are measured out with implementation using eight threads on an Intel Core i7–2600 processor, and that of the `bonsai` code (Bédorf et al., 2012), are on NVIDIA GeForce GTX 580.

ried out in a SIMD manner. The performance with multi-thread parallelization is almost proportional to the number of threads up to four threads. If the HTT is activated, the performance with eight threads is higher than that with four threads by a few percent.

The right panel of Figure 11 shows the performance of the AVX-accelerated implementation with eight threads for a various set of N_i and N_j. For $N_i \geq 64$, the performance is almost independent of N_i and N_j, and for $N_i = 16$ it is about half the performance with $N_i \geq 64$. This is again due to the overhead of copying j-particle data to the structure `Jpdata`, as is the case in the calculation of Newton's force. Such weak dependence of the performance on N_i and N_j are also preferable for the calculations of the forces in the PPPM and TreePM methods especially with the individual timestep scheme.

6. Summary

Using the AVX instructions, the new SIMD instructions of x86 processors, we develop a numerical library to accelerate the calculations of Newton's forces and arbitrarily shaped forces for N-body simulations. We implement the library by means of inline-assembly embedded in C-language with GCC extensions, which enables us to manually control the assignment of the YMM registers to computational data, and extract the full capability of a CPU core. In computing arbitrarily shaped forces, we refer to a look-up table, which is constructed with a novel scheme so that the binning is optimized to ensure good numerical accuracy of the computed forces while its size is kept small enough to avoid cache misses.

The performance of the version for Newton's forces reaches 2×10^9 interactions per second with a single thread, which is about 2 times and 20 times higher than those of

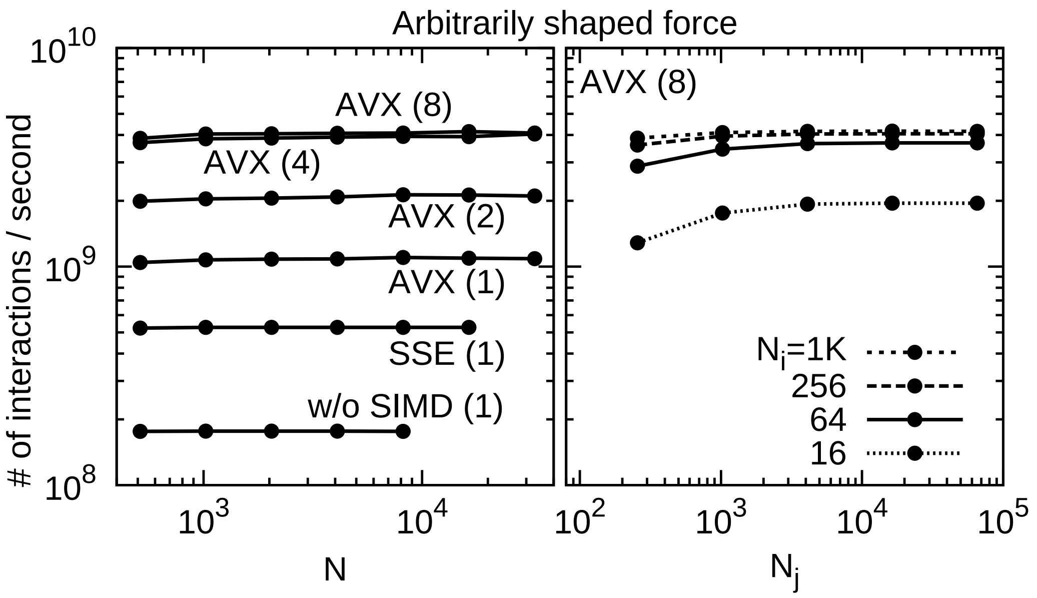


Fig. 11. (Left) Performance of calculations of arbitrarily shaped forces as a function of the number of particles, N. The vertical axis shows the number of interactions calculated per second. The numbers in parentheses beside indicate the number of threads. (Right) Performance of the AVX-accelerated implementation for the calculations of arbitrarily shaped forces for a various set of the number of i- and j-particles, N_i and N_j, respectively. The forces are computed with the AVX instructions on eight threads.

the implementation with the SSE instructions and without any explicit use of SIMD instructions, respectively. The use of the fast inverse-square-root instruction is a key ingredient of the improvement of the performance in the implementation with the SSE and AVX instructions. The performance of the version for arbitrarily shaped forces is 2 and 6 times higher than those implemented with the SSE instructions and without any explicit use of the SIMD instructions. Furthermore, our implementation supports the thread parallelization on a multi-core processor with the `OpenMP` programming interface, and has a good scalability regardless of the number of particles.

While the performance of our implementation using the AVX instructions is moderate compared with that of the GPU-accelerated implementation, the most remarkable advantage of our implementation is the fact that the performance has much weaker dependence on the numbers of i- and j-particles than that of the GPU-accelerated implementation. This feature is also the case for the calculation of the arbitrarily shaped force, and can be explained by the relatively large overhead of the data transfer between GPUs and main memory of their host computers. In actual calculations of forces with popular N-body solvers such as the Tree-method and the TreePM-method combined with the individual timestep scheme, the numbers of i- and j-particles cannot be always large enough to extract the full capability of GPUs. In that sense, our implementation is more suitable in accelerating the calculations of forces using the Tree- and TreePM-methods.

Another advantage of our implementation is its portability. With this library, we can carry out collisionless N-body simulations with a good performance even on supercomputer systems without any GPU-based accelerators. Note that massively parallel systems with GPU-based accelerators, at least currently, are not ubiquitous. Even on processors other than the x86 architecture, most of them supports similar SIMD instruction sets (e.g. Vector Multimedia Extension on IBM Power series, and HPC-ACE on SPARC64 VIIIfx, etc.) Our library can be ported to these processors with some acceptable efforts.

Finally let us mention the possible future improvement of our implementation. Fused Multiply-Add (FMA) instructions which have already been implemented in the "Bulldozer" CPU family by AMD Corporation, and is scheduled to be introduced in the "Haswell" processor by Intel Corporation in 2013. The use of the FMA instructions will improve the performance and accuracy of the calculations of forces to some extent. The numerical library "Phantom-GRAPE" developed in this work is publicly available at `http://code.google.com/p/phantom-grape/`.

Acknowledgment

We thank Dr. Takayuki Saitoh for valuable comments on this work. A. Tanikawa thanks Yohei Miki and Go Ogiya for fruitful discussion on GPU. Numerical simulations have been performed with computational facilities at the Center for Computational Sciences in University of Tsukuba. This work was supported by Scientific Research for Challenging Exploratory Research (21654026), Grant-in-Aid for Young Scientists (start-up: 21840015), the FIRST project based on the Grants-in-Aid for Specially Promoted Research by MEXT (16002003), and Grant-in-Aid for Scientific Research (S) by JSPS (20224002). K. Nitadori and T.

Okamoto acknowledge financial support by MEXT HPCI STRATEGIC PROGRAM.

References

Barnes,J., & Hut,P. 1986, Nature, 324, 446

Bédorf,J., Gaburov,E., & Portegies Zwart,S. 2012, Journal of Computational Physics, 231, 2825

Brieu, P.P., Summers, F.J., Ostriker, J.P. 1995, ApJ, 453, 566

Fukushige, T., Makino, J., Kawai, A. 2005, PASJ, 57, 1009

Gaburov, E., Harfst, S., & Portegies Zwart, S. 2009, NewA, 14, 630

Hamada, T., Iitaka, T. 2007, submitted, arXiv:astro-ph/0703100

Harfst, S., Gualandris, A., Merritt, D., Spurzem, R., Portegies Zwart, S., & Berczik, P. 2007, New Astronomy, 12, 357

Hockney, R.W., Eastwood, J.W. 1981, Computer Simulation Using Particles (New York: McGraw-Hill).

Ishiyama, T., Fukushige, T., & Makino, J. 2008, PASJ, 60, 13

Ishiyama, T., Fukushige, T., & Makino, J. 2009, ApJ, 696, 2115

Ishiyama, T., Fukushige, T., & Makino, J. 2009, PASJ, 61, 1319

Ishiyama, T., Makino, J., & Ebisuzaki, T. 2010, ApJ, 723, 195

Ishiyama, T., Makino, J., Portegies Zwart, S., Groen, D., Nitadori, K., Rieder, S., de Laat, C., McMillan, S., Hiraki, K., & Harfst, S. 2011, submitted, arXiv1101.2020

Johnson, V., & Aarseth, S. 2006, ASPC, 351, 165

Kawai, A., Fukushige, T., Makino, J., & Taiji, M. 2000, PASJ, 52, 659

Kawai, A., & Makino 2001, ApJL, 550, 143

Kawai, A., Makino, J., Ebisuzaki, T. 2004, ApJS, 151, 13.

Makino, J. 1991, PASJ, 43, 621.

Makino, J., & Aarseth, S. 1992, PASJ, 44, 141

Makino, J., Fukushige, T., Koga, M., Namura, K. 2003, PASJ, 55, 1163

Nitadori, K., Makino, J., & Hut, P. 2006, NewA, 12, 169

Oshino, S., Funato, Y., & Makino, J. 2011, PASJ, 63, 881

Portegies Zwart, S., Belleman, R. & Geldof, P. 2007, New Astronomy, 12, 641

Portegies Zwart, S., McMillan, S., Groen, D., Gualandris, A., Sipior, M., & Vermin, W. 2008, New Astronomy, 13, 285

Saitoh, T. R., Daisaka, H., Kokubo, E., Makino, J., Okamoto, T., Tomisaka, K., Wada, K. & Yoshida, N. 2008, PASJ, 60, 667

Saitoh, T. R., Daisaka, H., Kokubo, E., Makino, J., Okamoto, T., Tomisaka, K., Wada, K., & Yoshida, N. 2009, PASJ, 61, 481

Sugimoto, D., Chikada, Y., Makino, J., Ito, T., Ebisuzaki, T., Umemura, M. 1990. Nature 345, 33.

Tanikawa, A., & Fukushige, T. 2009, PASJ, 61, 721

Tanikawa, A., Yoshikawa, K., Okamoto, T., & Nitadori, K. 2012, New Astronomy, 17, 82 (paper I)

Xu, G., 1995. ApJS, 98 355

Yoshikawa, K., Fukushige, T. 2005. PASJ 57, 849.